

A Test Automation Framework for Mercury

Peter Biener, François Degraeve*, and Wim Vanhoof

Faculty of Computer Science, University of Namur, Belgium

Abstract This paper presents a test automation framework for Mercury programs. We developed a method that generates runnable Mercury code from a formalized test suite, and which code provides a report on execution about the success of test cases. We also developed a coverage tool for the framework, which identifies and provide a visualization of the reached parts of the program when executing a given test suite.

1 Introduction

Testing is today’s most commonly used method for finding defects and increase quality of software. It has been in focus for many years, since correcting defects is often the most substantial part of the software development budget [5]. When developing business critical systems, it is particularly important to detect bugs in time. Additionally, in this sector the release process is much stricter than usually, so detecting a bug only at a later phase of the release process can postpone the release date by months or years.

While the fact that the system under test successfully passes a large number of tests does not prove correctness of the software, it nevertheless increases confidence in its correctness and reliability [8].

In testing terminology, a *test case* for a software component refers to the combination of a single test input and the expected result. A *test suite* refers to a collection of individual test cases. Evaluating a test suite is a process that can be automated by using a tool that runs the software component that is being tested once for each test input, compares the actual result with the expected result and reports those test cases that failed during the test; the most well-known example of such a tool is JUnit for Java [7]. Such an automation framework allows the repeated evaluation of a test suite, for example to perform so-called regression testing – in order to detect errors that are introduced by changing code that was previously working fine.

There exist several test frameworks for declarative programming languages. For example, Prolog Unit Tests [14] – an integrated test framework for SWI-Prolog –, the basic `test_util` library for ECLiPSe Prolog, and HUnit [6] for Haskell. The target of this work is to develop a test automation framework for the Mercury language, which – to the best of our knowledge – has no such tool available yet. In our work we follow some principles from the mentioned tools, though we cannot port any of those tools directly to Mercury because of language

* Supported by a grant FRiA - Belgium.

specialities. For example, the strict type- and mode-checking mechanisms make it difficult to adopt most of the methods used in Prolog, even if we can of course re-use some of the ideas in the design phase.

The main goal of an integrated test framework is to interpret a previously formalized test suite, execute the independent test cases, and finally produce a report about which test cases failed and why. However, a test framework can also have additional features, which are not necessarily needed for the testing itself; a module could for example interact with an integrated debugger in order to try to identify the code fragment that caused the failure of a given test case [3]. It can also provide a coverage tool, i.e. a tool which is able to produce a measure describing the degree to which the source code of the program has been exercised during the execution of a given test suite.

This measure is performed with respect to one or more *coverage criteria* [15]. Among the most used coverage criteria, the *procedure* coverage criterion aims at verifying if every procedure is called, while the *entry/exit* coverage criterion is similar to the latter but takes into account the success or failure of a procedure's execution – which makes it particularly appropriate when testing logic programs. The *statement* coverage [10] criterion is relatively simple as it measures if every statement or instruction is reached during execution, whereas *branch* (or block) coverage criterion requires every condition of if-then-else statements to be evaluated to every possible value (usually true and false) (a variant of this criteria is used in the Emma coverage tool for Java [11]). The most general coverage criterion is the *path* coverage criterion, which requires every possible route through the code to be executed. Note that full path coverage is in general impossible; indeed, on the one hand loop constructs can result in an infinite number of paths, on the other hand the program under test can contain unreachable code fragments.

In what follows, we first introduce some background knowledge about the Mercury language (Section 2), then we present the unit testing tool and the interesting characteristics of its implementation (Section 3). In Section 4 we present the details of the coverage tool, then we show and discuss the results of the evaluation of the prototype (Section 5).

2 Preliminaries

Mercury is a statically typed logic programming language [12]. Its type system is based on polymorphic many-sorted logic and is essentially equivalent to the Mycroft-O'Keefe type system [9]. A type definition defines a possibly polymorphic type by giving the set of function symbols to which variables of that type may be bound as well as the type of the arguments of those functors [12]. Take for example the definition of the well known polymorphic type *list*(*T*):

```
:- type list(T) ---> [] ; [T|list(T)].
```

According to this definition, if T is a type representing a given set of terms, values of type $list(T)$ are either the empty list `[]` or a term $[t_1|t_2]$ where t_1 is of type T and t_2 of type $list(T)$.

In addition to these so-called *algebraic types*, Mercury defines a number of primitive types that are builtin in the system. Among these are the *numeric types* `int` (integers) and `float` (floating point numbers). Mercury programs are statically typed: the programmer declares the type of every argument of every predicate and from this information the compiler infers the type of every local variable and verifies that the program is well-typed.

In addition, the Mercury *mode system* describes how the instantiation of a variable changes over the execution of a goal. Each predicate argument is classified as either input (ground term before and after a call) or output (free variable at the time of the call that will be instantiated to a ground term). A predicate may have more than one mode, each mode representing a particular usage of the predicate. Each such mode is called a *procedure* in Mercury terminology. Each procedure has a declared (or inferred) *determinism* stating the number of solutions it can generate and whether it can fail. Determinisms supported by Mercury include `det` (a call to the procedure will succeed exactly once), `semidet` (a call will either succeed once or fail), `multi` (a call will generate one or more solutions), and `nondet` (a call can either fail or generate one or more solutions)¹. Let us consider for example the definition of the well-known `append/3` and `member/2` predicates. We provide two mode declarations for each predicate, reflecting their most common usages:

```
:- pred append (list(T), list(T), list(T)).
:- mode append(in, in, out) is det.
:- mode append(out, out, in) is multi.

append([], Y, Y).
append([E|Es], Y, [E|Zs]) :- append(Es, Y, Zs).

:- pred member(T, list(T)).
:- mode member(in, in) is semidet.
:- mode member(out, in) is nondet.

member(X, [X|_]).
member(X, [_|T]) :- not (X=Y), member(X, T).
```

For `append/3`, either the first two arguments are input and the third one is output in which case the call is deterministic (it will succeed exactly once), or the third argument is input and the first two are output in which case the call may generate multiple solutions. Note that no call to `append/3` in either of these modes can fail. For `member/2`, either both arguments are input and the call will either succeed once or fail, or only the second argument is input, in which case

¹ There exist other modes and determinisms but they are outside the scope of this paper; we refer to [12] for details

the call can fail, or generate one or more solutions. Note that unlike Prolog, Mercury doesn't handle partially instantiated data structures.

3 Unit testing tool for Mercury

The goal of this work is to create a framework for Mercury that lets the user define test cases through a simple language, and from that point on, automatically performs the whole testing process. In our implementation, the framework has two independent modules: one is responsible for executing a test suite, the other is a coverage tool, which is presented later in this paper. The two modules are loosely connected, each one being usable without the other.

The generation process is completely independent from the tested code: one can write test cases without any knowledge of the source code, so the tool is even usable for test-driven development.

A schematic diagram of the testing process is shown on Figure 1. The test cases are contained in the test suite file. The other (optional) input of the tool is a renaming information file, which is generated by the coverage tool. Its usage is explained in Section 4. In general, a formal test case representation consists of a unique test case, i.e. the combination of a code fragment to be executed together with one or more assertions on the expected results. In our implementation, a test case is a triple, written as $\text{test}(t, c, a)$, where t is the name of the test case (a Mercury string), c is a Mercury code fragment represented as a list of atoms, and a is a list of assertions. An assertion can be either a condition on the variables of c , or a specification of the expected behaviour of the execution. For the latter, the test framework provides three options: **succeed**, **fail** or **exception**.

Let's examine a simple example of the syntax of a test case:

Example 1.

```
test(t1, [reverse([1,2],L)], [true(L=[2,1])]).
```

`t1` is the name that will be used to refer to the test case in the report generated by the testing tool. The code fragment to test contains only one goal (a call to the list reverse predicate), while the only assertion is a condition verifying whether the value computed for `L` is indeed the result of reversing the list `[1,2]`.

3.1 Determinism

If only the features mentioned above are used then execution of the test code is limited to the first solution, even if the predicate under concern has possibly multiple solutions. In the latter cases, all the solutions but the first one are dropped. Nevertheless, more extensive examination of **multi** or **nondet** predicates is also possible. In order to achieve this, the following conditions can be used in the assertions part:

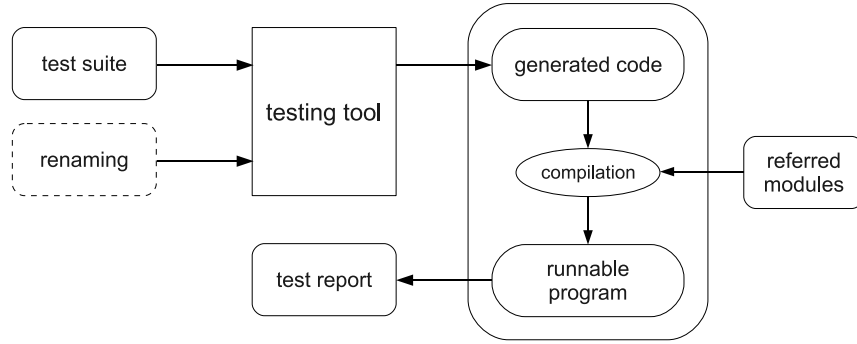


Figure 1. Testing framework

true(C) Simple execution of C. Only a single instantiation of the assertion C is verified, namely the one obtained from the first answer returned by the tested code in the test case.

some_true(C) The given condition holds for at least one solution of the tested code.

all_true(C) The given condition holds for all solutions.

true(N, C) The given condition holds for the Nth solution².

solutions_cardinality(N) N is equal to the number of solutions. N can be either a variable or a constant.

type(V, T) User defined type information of a variable.

limit(N) Limits the execution to N solutions. This can be useful when testing predicates with a large number of solutions.

Example 2 shows the usage of some of these conditions.

Example 2.

```

test(t2, [member(X,[1,3,4,2])],
        [limit(2),some_true(X>1)]).
test(t3, [member(X,[1,2,3,4])],
        [solutions_cardinality(N),true(N>3),
         all_true(X<5)]).

```

The semantics of the assertions in **t2** is “there is a solution among the first 2 that is bigger than 1”, while the meaning of the assertions in **t3** is “there are at least 3 solutions, and all of the solutions are less than 5”.

Usage of input/output operations is not permitted in the assertions part, but they are allowed in the tested code fragment under the condition that all the goals are deterministic. In practice it means that there are two different

² This is definitely not a pure declarative condition, since the result may depend on the order of results, and thus on the Mercury implementation.

execution modes of the test tool: “multi” and “IO”. With “multi”, testing multi-solution predicates is possible, but usage of IO is not. With “IO”, it is the other way round. This execution mode of the tool can be chosen by a command line option.

3.2 Implementation details

Figure 2 shows the generated code for the test case `t1` defined in Example 1. Since the expected behaviour is specified as success, if the `reverse/2` predicate fails, the result of the test case will be “failed because of failure (instead of success)”.

```
...
testcase(t1, Result) :-
    (if
        reverse([1,2], L)
    then
        (if
            L = [2,1]
        then
            Result = succeeded
        else
            Result = condition_failed
        )
    else
        Result = failed(failure)
    ).
```

Figure 2. Generated code (det)

Testing multi-solution predicates needs some considerations. Our implementation uses Mercury’s `solutions` library for handling predicates that can succeed more than once. However, this library has an important restriction, namely that the given predicate can have only one output argument. An easy solution to this problem is to wrap all the output variables into a compound term, then unwrap the variables after execution of the code and then perform the checks of the assertions part. Unfortunately, for the generation of this compound type declaration, the type of each output variable should be known. The need of type analysis could strongly limit the usability of the tool since all the sources of used modules should be known in that case. This is usually not feasible, especially in case of built-in modules. The workaround we developed is to use the type analysis facility of the compiler itself. It is possible with the `univ` library, which allows to wrap any Mercury type into a universal type. For the unwrap operation, the compiler must know the type of the wrapped object. Usually, it can be

inferred from the assertions, but if not, we have to give the type manually, as a help to the compiler. Example 3 shows the usage of this feature.

Example 3.

```
test(t4, [append(L1,L2,[1,2,3])], [type(L2,list(int)),
    some_true((L1=[1,2],length(L2,1)))]).
test(t5, [append(L1,L2,[1,2,3])],
    [some_true((L1=[1,2],L2=[3]))]).
```

The tested code fragment is the same in both test cases: the `(out,out,in)` mode of `append/3`. In `t4`, the compiler can infer the type of `L1`, but the type of `L2` must be given explicitly. In the other test case, the compiler doesn't need any complementary information. Notice that if there is no more than one common variable between the two code parts, then no wrapping is used, and thus no type information needs to be provided.

Generation of code for the `some_true/1`, `all_true/1` and `true/2` conditions is based on the same principle. Unwrap instructions of output variables are appended before the given condition if necessary, then this code fragment – a meta-predicate – is called in an appropriate way. For example in the case of `true/2`, after selecting the required solution from the list, the constructed meta-predicate is called simply using the `call/2` predicate. The optional solution number limitation is implemented with the help of `do_while` predicate in the `solutions` library.

Figure 3 shows the generated code for the test case `t4`, where we can see the declaration for the generated type. The two lines just after the call to `append/3` wrap the output variables into a single compound term. The reverse operation is performed by the two lines just before the assertions. The combination of the latter together with the assertions themselves constitute the body of a meta-predicate. This meta-predicate must succeed for at least one solution for the test case to be considered as successful.

3.3 Handling exceptions

By default, every exception thrown by either the tested code or some of the assertions is caught by the framework. This is necessary, otherwise it could interrupt the execution of a large test suite with possibly a very large number of test cases. In special cases, the expected result can be exception too, which is a feature the framework can handle with a small restriction: exceptions cannot be distinguished by their origin, one thrown by an assertion is handled in the same way as if it had been thrown by the tested code. Currently it is impossible to make assertions on the exception itself, the only thing that can be declared in the assertion part is “the code must throw an exception”.

Nevertheless, it can happen that exceptions need to be left uncaught, especially when the user wants to know the exact source of an exception, usually to know where to find a given bug. If the exception is caught, the result will only

```

...
:- type t4_type ---> t4_t(univ, list(int)).
testcase(t4, Result) :-
  solutions( ((pred (IF1 :: out)) is nondet :-
    append(L1, L2, [1,2,3]),
    type_to_univ(L1, L1_U),
    IF1 = t4_t(L1_U, L2)
  ), Vs),
  (if
    some_true( ((pred (IF2 :: in)) is nondet :-
      IF2 = t4_t(L1_U, L2),
      det_univ_to_type(L1_U, L1),
      L1 = [1,2], length(L2,1)
    ), Vs)
  then
    Result = succeeded
  else
    Result = condition_failed
  ).

```

Figure 3. Generated code (nondet)

be “the test case threw an exception”, but the real cause remains hidden. To help to identify these problems, exception handling mechanism can be entirely switched off by a command line switch, so in that “debug” mode, the details of the problem become observable.

4 Coverage tool

This tool is a complementary module for the base framework that helps to detect parts of the tested code which are uncovered by a given set of test cases. Logic programming languages have a few peculiarities that must be taken into account when constructing a coverage tool. The most important of these is non-determinism, namely that statements can fail and/or have multiple solutions. Most coverage tools for declarative languages transform the original program to an instrumented code and place some kind of execution counters before and after calls. This enables tracing calls to and exits from procedures. The counters are usually stored in a non-declarative way, like in the Haskell Program Coverage tool [4], which records every increment into a file. This is unavoidable in case of logic programming languages, since after a backtracking, all changes made on pure declarative variables would be revoked. The same principle is used in the **coverage** library for ECLiPSe Prolog, the output of which is a simple HTML page, where the counter values are shown between the goals under concern.

Our tool follows the same principles as these tools, but needs to deal with some particularities of the Mercury language. The most notable one is the mode-

reordering mechanism of the Mercury compiler, as it strongly affects coverage. Another issue that is worth mentioning is the way that the Mercury compiler treats switches. In the next section, we expose these different issues and present the solutions we developed.

4.1 Implementation

In the remaining, we assume that the programs are well-moded; this condition is checked during compilation by the Mercury compiler. The latter also re-orders the goals in such a way that they are executed from left to right. Since different modes usually imply different orders, multi-moded predicates must be transformed into several different procedures.

Our implementation transforms the examined code into an instrumented one, compiles it and executes it in order to log execution information; the process of coverage measuring is shown in Figure 4. The base idea of the transformation is to add counters in the code, implemented by logging calls that write unique identifiers into a log file. The counters are placed with respect to the labelled superhomogenous form syntax defined in [2], with minor simplifications:

Definition 1. *The syntax of a program in labelled superhomogenous form is defined as follows:*

$$\begin{aligned} LProc & ::= p(X_1, \dots, X_k) \text{ :- } LConj. \\ LConj \ C & ::= {}_lG_l' \mid {}_lG, C \\ LDisj \ D & ::= C; C'' \mid D; C \\ LGoal \ G & ::= A \mid D \mid \text{not}(C) \mid \text{if } C \text{ then } C' \text{ else } C'' \\ Atom \ A & ::= X = Y \mid X = f(Y_1, \dots, Y_n) \mid p(X_1, \dots, X_n) \end{aligned}$$

A counter is assigned to each label l , denoted by $\text{counter}(l)$. Basically this means that counters are inserted into every possible place between goals.

The first step of the transformation process is to get the code to be in superhomogeneous form. A part of this process can be achieved by the compiler (goals reordering, duplication of predicates with multiple modes); however all the multi-moded predicates need to be renamed, in such a way that every procedure is associated to a unique name. Every call to the procedures must therefore be renamed consequently; this can be done thanks to a simplified mode analysis, following the instantiations of variables throughout the code. The new name assignments are saved into a file, in order to provide names mapping information to the user at the end of the process.

4.2 Switches vs disjunctions

The second step of the transformation is the addition of logger calls between goals of the code; these calls reify incrementing operations on counters: $\text{log}(l) \equiv \{\text{counter}(l) := \text{counter}(l) + 1\}$. Unfortunately, this step can render the program not compilable if it contains *switches*. A switch is a special disjunction – with

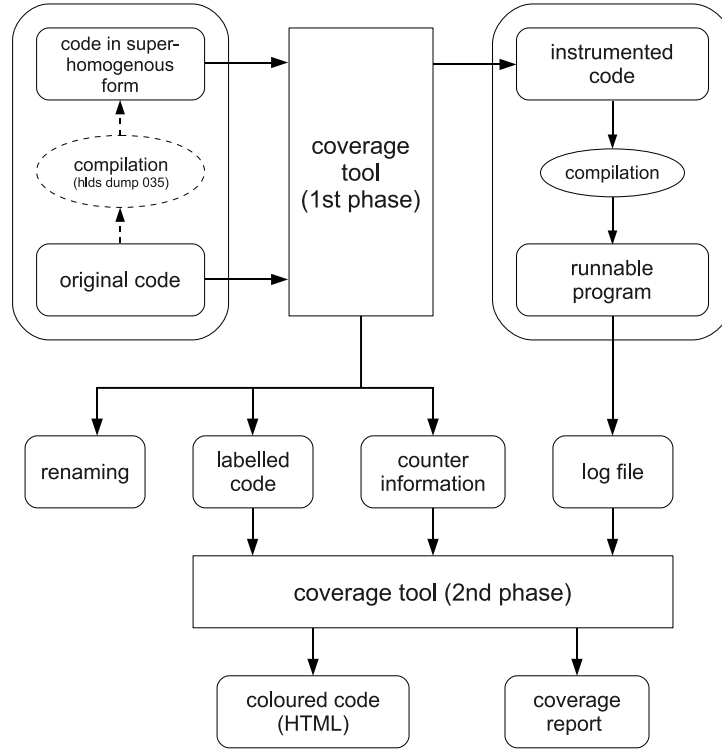


Figure 4. Coverage tool

nothing visually distinguishing it from a “regular” disjunction \vee , in which “each disjunct has near its start a unification that tests the same bound variable against a different function symbol” [13]. In the remaining, we call such unifications the *switch conditions*. In a single switch, the switch conditions are exclusive from each other; that allows the compiler to consider the switch as being deterministic or semi-deterministic – depending on whether every possible condition value is covered – whereas regular disjunctions are, in general, non- or multi-deterministic. Switches can be nested into each other and if they test the same variables, they are likely to be treated as a single switch.

The reason switches are considered as particular structures is to allow the compiler to perform a determinism analysis; no regular disjunction is allowed in predicates declared as *det* or *semidet*. Only unifications can precede switch conditions in the different disjuncts; if not, the compiler is not able to detect the switch conditions, and therefore considers the disjunction under concern as a regular (*nondet* or *multi*) disjunction. When logger calls are inserted at the beginning of a disjunct, a switch will therefore be considered as a regular disjunction. That can cause the compilation to fail if the enclosing predicate is (semi-)deterministic. The example shown at the left side of Figure 5 is extracted

<pre> ... , (X = f, p(Out) ; Y = X, (Y = g, Intermediate = 42 ; Z = Y, Z = h(Arg), q(Arg, Intermediate)), r(Intermediate, Out)), ... </pre>	<pre> (log("label_1"), X = f, log("label_2"), p(Out), log("label_3") ; log("label_4"), Y = X, log("label_5"), (log("label_6"), Y = g, log("label_7"), Intermediate = 42, log("label_8") ; log("label_9"), Z = Y, log("label_10"), Z = h(Arg), log("label_11"), q(Arg, Intermediate), log("label_12")), log("label_13"), r(Intermediate, Out), log("label_14")) </pre>
---	---

Figure 5. Naive instrumentation of a switch

from the Mercury reference manual: it is a switch on X , provided X is ground. On the right side, there is the “naively” instrumented version with the logger calls; this instrumented code would cause the program to fail at compiling if it is used in a (semi-) deterministic predicate.

The solution we developed is to replace, in a disjunct, logger calls before each unification at the beginning of a disjunct by a single logger predicate call *after* the unifications – just before the first predicate call occurring in the disjunct. This single logger call should update all the counters in order to reflect success or failure of all the preceding disjuncts. Unfortunately, this solution has a drawback: if one of the unifications preceding the logger call fails during execution, it is not possible to know which one it was (since no counter was placed between them) and then the coverage information is incomplete. However, we can recover this missing information by performing a small analysis since the values of the

variables involved in these unifications are known before entering the disjunction. The algorithm that determines which counter needs to be updated at which point is presented below (its basic steps are shown on Figure 7).

We model a switch under the form of a tree, since they can be nested into each other. Nodes of the tree are the labels of the labelled superhomogenous form of the program, while its edges are the unifications between the labels. All the statements after the first predicate call of each disjunct are dropped from the model, so the leaves of the tree are the labels preceding the first predicate call in each disjunct. Complex statements, like conditional structures, etc. are treated as if they were predicate calls, and are thus also dropped from the model. If there are only unifications in a disjunct, then the leaf of the corresponding branch is the last label of the disjunct. We can define a *simplified execution path* for a switch as a sequence of labels, which is the output of a depth-first search in the corresponding model graph. The model of the example of Figure 5 is shown in Figure 6, while the corresponding simplified execution path is (1,2,4,5,6,7,8,9,10,11).

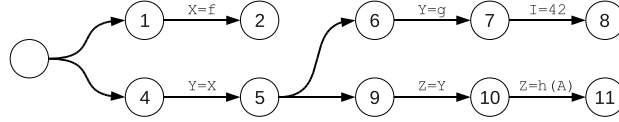


Figure 6. Switch execution graph

Before executing a switch, all edges of its model graph are examined. If a unification succeeds, then its successor node is marked, otherwise this part of the tree is left out from further examination. The first node of each branch is also marked, since they are not assigned to any unification. After this step, the nodes that are not marked correspond to program points that are not reached on the examined program state, and thus no counter update is applied there. The marked nodes are visited using a depth-first search; when a leaf is reached, the sequence of marked nodes encountered on the path up to this leaf is stored and associated to the leaf's label. This process is repeated starting from the next unvisited marked node until no marked node is left unvisited.

Thanks to this method, a simplified execution path is split into sequences, whose last elements are mapped to program points that are reached on the examined program state. These are the sequences of labels that must be written out to the log file when the execution reaches the given label.

Using the example from Figure 6 again, and assuming that every unification was successful (which is of course impossible, since the value of X cannot be f , g and $h(\text{Arg})$ at the same time), the sequences assigned to leaves would be (1,2) for leaf 2, (4,5,6,7,8) for leaf 8, and (9,10,11) for leaf 11. If the unification $X=f$ between the first two labels fails, then the assignments will be (1,4,5,6,7,8) for leaf 8, and (9,10,11) for leaf 11. The original logging statements can be removed

from the code, and only one batch logging statement is needed at the “leaves” of the switch (in each successful branch).

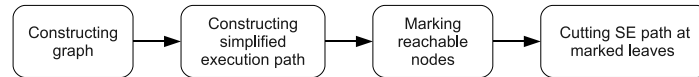


Figure 7. Switch transformation

The last issue remaining using that technique is due to the last segment of the simplified execution path when the last node is not marked. It means that the last branch (or the last few branches) fails, and in this case we have no information about which unifications were executed. We decided to log these entries at the same time with the previous batch logging action, or if there is no successful branch, log them before the execution of the switch. For example, if the failing unifications are $X=f$ and $Z=h(\text{Arg})$, then the only batch will be (1,4,5,6,7,8,9,10) associated to leaf 8.

4.3 Connection with the base framework

Once the code has been successfully instrumented, using the coverage tool as a part of the testing framework is easy: in the test suite file, we simply refer to this transformed code instead of the original one. For convenience, it isn’t necessary to rename multimode predicates in test cases, it is possible to pass the renaming information file – which is generated by the instrumentation step – to the test automation module. With this feature, the new predicates will be called on test execution.

4.4 Statistics and report

The direct output of executing instrumented code is a log file, which contains information about reached program points. However this information is not easily readable; it should appear under a more “user-friendly” form in order to be useful. For this purpose there is a second execution phase in our implementation. It makes use of different information sources: the labelled source file (the instrumented code just before switch transformation – possibly not compilable) and a file containing meta-information about counters. This meta information consists of the correspondences between pairs of counters and the points of interest (simple goals or complex structures such as disjunctions). One additional pair of counters is added for each predicate; they are the first and the last counters of the predicate. It can be used to evaluate predicate or procedure coverage. The log file is generated at execution time, while the other two are created at the same time as the instrumentation.

Currently, the output of this coverage tool is a html file containing the source code in which colours have been added. Only goal coverage is taken into account

for the moment – this can be easily extended using information provided by the log file. Three coverage degrees are distinguished:

covered (green) execution of the goal was successful (at least once)
partially covered (yellow) the goal was executed, but never succeeded
not covered (red) the goal was not executed

The tool also produces a detailed report which enumerates all the pairs of counters, and gives the coverage degree of the corresponding goal. Although it is less visual than the html rendering, this report contains more information, since it also gives the coverage degree of complex structures (disjunctions, switches) and predicates (procedure coverage).

5 Evaluation

Table 1 shows the results of a small evaluation of our tool. Three different properties were examined: for a given testsuite, we measure the size of the generated code, the time needed for its generation by our tool, and the execution overhead of the generated code compared to the execution time of a script that executes the testsuite in an ad-hoc way. The testsuites for the first three procedures were automatically generated by [2], while the testsuite for the last procedure (transpose) was a manually written testsuite containing matrices up to a size of 3x3.

Goals	Determinism	Test cases	Generated code size (lines)	Code generation (ms)	Execution (ns)	
					gross	net
member(in,in)	semidet	6	169	12	40	2
member(out,in)	nondet	4	189	12	40	11
bubblesort(in,out)	det	24	475	16	60	50
transpose(in,out)	det	11	288	12	40	8

Table 1. Performance of the testing tool

As one expects, the size and generation time of the code depends on the number and complexity of the given set of test cases. Although the execution time also depends on the complexity of the test cases, the evaluation shows a rather constant overhead for the execution of the testcode generated by our framework.

Table 2 illustrates the performance of the coverage tool. The examined properties are the size of the instrumented code compared to the size of the original source code, the time needed for instrumentation and the execution overhead caused by the transformation. The table also shows the execution times for both the instrumented and non-instrumented code. The tested procedures are the

same as those in Table 1 with the additional `filter_list`, the latter being a procedure from the code of the test framework itself that does list filtering by a given set of indices. The input parameters are chosen relatively large in order to produce measurable times (lists of a few hundred to few thousand elements).

Goals	Code size (lines)		Instrumentation (ms)	Execution (ms)	
	original	instrumented		original	instrumented
member(in,in)	19	124	32	13	2050
member(out,in)				42	4460
bubblesort(in,out)	38	177	53	11	7130
transpose(in,out)	58	340	96	3.3	1520
filter_list(in,in,out)	37	191	60	18	4760

Table 2. Performance of the coverage tool

Since counter update occurs between goals, the size of the instrumented code (in number of lines) is approximately twice the size of the original code. However, when the switch transformation is applied, additional lines are added for every switch test statement, but in any case the size of the instrumented code is limited to a few times the size of the original one. However, as can be seen from Table 2, the execution time overhead of the instrumented code can be significant. This can be explained in part by the overhead due to the logging operations, in part by the fact that the compiler no longer can perform a number of optimisations. Nevertheless, it should be noted that the execution overhead is only present when one is measuring test case coverage, and not when one is executing the testsuite.

6 Conclusion and Future Work

In this paper, we have presented a test evaluation framework for Mercury. The framework is implemented, and allows one to write and execute a test suite for a Mercury program, and to visualise the code that is (not) covered by the test suite. The implementation is published as open source software [1].

Although our implementation is usable for testing small to medium-size Mercury programs, some features are missing in order to make our tool a full-fledged testing tool for Mercury. In the current implementation, the coverage transformation is applied to one module at a time. Consequently, when measuring the coverage of a multi-module program, the generated renaming information files should be merged manually. Even with this workaround, incorrect results might be produced if two or more examined modules contain predicates having the same name. This is because the lightweight mode analysis that is performed by the tool is not sophisticated enough to choose the correct predicate from the

several candidates in different modules. Neither does the analysis handle partial instantiation of variables, a somewhat lesser problem given that the Mercury compiler itself has only limited support for these partially instantiated structures.

Also the framework definition itself could be extended. Additional conditions could be introduced for the assertions part of test cases allowing, for example, to state conditions on the relation between the different solutions of a predicate. One example would be a condition that checks whether the solutions returned by a call are in ascending order. Further improvements include changing the output format of the framework, for example to xml, in order to make postprocessing of the report easier. A more fundamental improvement would be the introduction of more sophisticated coverage levels into the tool. These improvements are subject to ongoing and further work.

References

1. Peter Biener. Mercury test framework. <http://sourceforge.net/projects/mercurystest>, 2010.
2. François Degraeve, Tom Schrijvers, and Wim Vanhoof. Automatic generation of test inputs for Mercury. In Michael Hanus, editor, *LOPSTR*, volume 5438 of *Lecture Notes in Computer Science*, pages 71–86. Springer, 2008.
3. M. Ducassé and A.-M. Emde. A review of automated debugging systems: knowledge, strategies and techniques. In *ICSE '88: Proceedings of the 10th international conference on Software engineering*, pages 162–171, Los Alamitos, CA, USA, 1988. IEEE Computer Society Press.
4. Andy Gill and Colin Runciman. Haskell program coverage. In Gabriele Keller, editor, *Haskell*, pages 1–12. ACM, 2007.
5. R. Glass. *Software runaways: lessons learned from massive software project failures*. Prentice Hall, 1997.
6. Dean Herington. *HUnit User's Guide*. <http://hunit.sourceforge.net/HUnit-1.0/Guide.html>, 2002.
7. A. Hunt and D. Thomas. Pragmatic unit testing in java with junit. Pragmatic Bookshelf, 2003.
8. C. Kaner, J. Falk, and H. Q. Nguyen. *Testing computer software*. John Wiley and Sons, 1993.
9. A. Mycroft and R.A. O’Keefe. A polymorphic type system for Prolog. *Artificial Intelligence*, 23:295–307, 1984.
10. Glenford J. Myers. *The Art of Software Testing, Second Edition*. Wiley, 2004.
11. V. Roubtsov. *EMMA: a free Java code coverage tool, Reference Manual*. <http://emma.sourceforge.net/reference/reference.pdf>.
12. Z. Somogyi, H. Henderson, and T. Conway. The execution algorithm of Mercury, an efficient purely declarative logic programming language. *Journal of Logic Programming*, 29(4), 1997.
13. University of Melbourne, http://www.cs.mu.oz.au/research/mercury/information/doc-release/reference_manual.pdf. *Mercury Language Reference Manual*.
14. Jan Wielemaker. *Prolog Unit Tests. Manual*. <http://www.swi-prolog.org/packages/plunit.html>, 2006.
15. H. Zhu, P. Hall, and J. May. Software unit test coverage and adequacy. *ACM Computing Surveys*, 29(4), 1997.